Inteligência Artificial - 2016.2
Othello

Professor João Carlos
DCC/IM/UFRJ

18 de dezembro de 2016

Componentes do grupo:

Claudia Tomaz
Eduardo Costa

1. Introdução

Neste trabalho, foi pedido para que fosse implementado um agente inteligente para jogar o jogo Othello (também conhecido como Reversi), a partir de um código do jogo já implementado. Foram implementados dois agentes, e um deles foi usado na competição em sala de aula, no dia 15 de dezembro de 2016, jogando contra os agentes dos outros grupos.

2. Agente 1: Guloso

Nesse primeiro agente, foi implementado um algoritmo guloso, que sempre escolhe a jogada que "come" mais peças do inimigo.

Este agente está disponível em:
https://github.com/edunmc/TabuleiroOthello/blob/master/models/players/claudia_eduardo_guloso_player.py

Para cada jogada possível, o algoritmo calcula a quantidade de peças do inimigo no "caminho", e escolhe a que tiver mais. Nos casos de empate (mais de uma jogada com o maior número de peças para comer), para não se tornar muito determinístico, é realizada uma escolha aleatória entre os melhores.

Como esperado, o resultado dos testes de jogos com outros agentes não foi muito bom.

3. Agente 2: Minimax

Implementação do algoritmo minimax, com corte alfa-beta. Foi o agente usado no campeonato.

Este agente está disponível em:
https://github.com/edunmc/TabuleiroOthello/blob/master/models/players/claudia_eduardo_minimax_player.py

3.1. Função de avaliação

A função de avaliação utilizada é mostrada a seguir. "Canto" se refere a uma das 4 casas nos extremos do tabuleiro, e "semi-canto" se refere a uma das 3 casas que cercam cada canto.

Cada casa tem um valor, e é feito um somatório do valor de todas as casas com peças:

- Nos casos específicos seguintes, são somados os valores abaixo:
  - peça minha no canto: +10000
  - peça minha num semi-canto: -100000
  - peça minha numa parede: +5000
  - peça do inimigo num canto: -100000
  - peça do inimigo num semi-canto: +10000
- Nos outros casos: (peças que não se encaixam nos casos específicos acima)
  - oposto da distância Manhattan (×(-1)) de cada peça minha até o canto mais próximo
  - distância Manhattan de cada peça do oponente até o canto mais próximo

Dessa forma, quanto maior o valor, melhor pra mim.

3.2. Profundidade máxima

A profundidade máxima da árvore de busca foi definida como 4. As funções **mini** e **maxi** do código possuem um argumento **prof_max**, que indica a profundidade máxima. Este limite é definido na chamada dessas funções:

```
def maxi(self, estado, prof_max, prof_atual, beta):
```

- **estado**: estado do tabuleiro (nó) a partir do qual vamos expandir a árvore
- **prof_max**: profundidade limite
- **prof_atual**: profundidade atual
- **beta**: valor do beta do **mini** que chamou esse **max**

E, analogamente, a função **mini**, com os mesmos argumentos, exceto o último, que é **alfa**, indicando o valor do **alfa** da **max** que a chamou.

3.3. Corte alfa-beta

O corte alfa-beta foi utilizado. As partes do código responsáveis estão mostradas abaixo:

O **alfa** de cada **max** é passado para o **mini** ao chamá-lo:

```
def mini(self, estado, prof_max, prof_atual, alfa):
```

Dentro da **mini**, ele é comparado com cada valor de filho do mini.

```
if valor < alfa:
  return (1000, None)
```

Se o valor de um dos filhos do **mini** (ou seja, **beta** <= esse valor) for menor do que o **alfa** do pai (**max** que chamou o **mini** atual), podemos parar este ramo por aqui (e não importa o que for retornado, já que quem definiu o **alfa** do pai é melhor).

Analogamente, temos o mesmo na função **maxi**:

O **beta** de cada **min** é passado para a **max** ao chamá-la:

```
def maxi(self, estado, prof_max, prof_atual, beta):
```

Dentro da **maxi**, ele é comparado com cada valor de filho do max.

```
if valor > beta:
  return (-1000, None)
```

Se o valor de um dos filhos do **max** (ou seja, **alfa** >= esse valor) for maior do que o **beta** do pai (**mini** que chamou o **max** atual), podemos parar este ramo por aqui (e não importa o que for retornado, já que quem definiu o **beta** do pai é melhor).

4. Referências

1. Winning Reversi with Monte Carlo Tree Search. Disponível em: <https://andysalerno.com/2016/03/Monte-Carlo-Reversi>. Acesso em: 2 de dezembro de 2016

2. Zolomon/reversi-ai: A text based python implementation of the Reversi game with an artificial intelligence as opponent. Disponível em: <https://github.com/Zolomon/reversi-ai>. Acesso em: 2 de dezembro de 2016.